Project Report

EE565 Information Theory and its Application to Big Data Sciences

Dhruv Parikh

2522583608

Prof. Urbashi Mitra

1. Encoding

For encoding, I used an extremely simple LZ 77 sliding window strategy. Effectively, the encoder starts from the original string, converts the string into a list, and initiates encoding it from the first entry of the list.

For my encoding purposes,

Window Size = 99,999

Maximum Match Length = 99,999

As is, I did modify the basic sliding window algorithm slightly to ensure increased efficiency.

The modifications were as follows:

1. I did not encode if Match Length was only 1

2. For match length 2 and 3, I encoded only under certain specific conditions, whereby the encoding guaranteed efficiency.

3. I did not encode flag bit for the case when there was a match (Flag bit = 1 in F, P, L was not encoded)

4. Rather than use a fixed length representation of the P and L quantities in cases with a match, or using a universal integer encoding strategy, I used a length representation strategy that specified the length of the P and L quantities.

To clarify things more precisely, consider the string:

ABC AAB 112

4122

In the first step, it is read into a content string. Then the content string is converted into a list, as follows:

[‘A’, ‘B’, ‘C’, ‘ ‘, ‘A’, ‘A’, ‘B’, ‘ ‘, ‘1’, ‘1’, ‘2’, ‘\n’, ‘4’, ‘1’, ‘2’, ‘2’]

Next, from this list, we start encoding. The encoder outputs a list, where modifications 1, 2 and 3 are used, as mentioned above. In effect, the output is:

[0, ‘A’, 0, ‘B’, 1, 3, 0, ‘C’, 7, 5]

(mock example, not actual encoded version of the above string)

Now, in effect, the 1, 3 represent P, L and the 7, 5 also represent P, L.

It is extremely clear that in our sequential encoding and decoding strategy, we do not need to encode the Flag = 1 bit for the strings that are encoded.

Additionally, it is important to see that P and L are in the set {1, 2, ……., 99999} and thus can never be 0. Thus, we will also know, sequentially, when a particular subsequent list entry belongs to the P, L pair or the 0, ‘char’ pair.

Now, a major issue with this strategy, on its own, was that if converted into a string and then converted into a .txt encoded file, while decoding, we cannot precisely figure out where characters of P start and end and where characters of L start and end.

Thus,

[1, 30, 3, 40] when converted to a string is '130340' and thus could mean a variety of P, L combinations other than the correct one.

To solve this issue, we modified the encoded list slightly (with modification 4) so that converting the list to a string didn't lead to any kind of ambiguity.

Since,

Window Size = Maximum Match Length = 99,999

The range of the length of P and L entities will be {1, 2, 3, 4, 5}

Thus consider the tuple (length P, length L). We have a total of 5*5 = 25 possible combinations in this tuple, and thus, can encode it effectively using the following strategy:

(1, 1) ---------- A

(1, 2) ---------- B

(1, 3) ---------- C

…… ..

…… ..

…… ..

(5, 5) ---------- Y

Thus, instead of using 2 bytes for encoding (length P, length L) for each (P, L), we end up using only 1 byte, by using the above encoding strategy.

Thus, (F, P, L) was converted to (P, L) then (C, P, L) where C = {A, B, C, …, Y} which points to a (length P, length L) pair.

Thus, as an example, consider,

[0, 'A', 0, 'B', 1, 2, 1, 3, 5,5]

This is encoded, finally, as:

[0, 'A', 0, 'B', 'B', 1, 2, 'C', 1, 3, 'Y', 5, 5]

The final string written to encoded text is,

0A0BB12C13Y55

Again, due to the sequential nature of our encoding/decoding strategy, not seeing a 0 automatically tells the decoder to expect a [C, P, L] string.

Finally, consider L = 1 case (match length 1) and consider [C, P, L]

[1 byte, at least 1 byte, 1 byte]

Thus, encoding it will use up at least 3 bytes while not encoding it will use up only 2 bytes, making not encoding it always more efficient.

For L = 2 and L = 3 cases, we compare the memory of the encoded and the unencoded strings and limit the range of P over which we encode to ensure optimality.

For L at least 4, it is always more optimal to encode.

This completes the discussion of our encoding strategy.

2. Decoding

Decoding the string is so straightforward that it shouldn't even warrant a discussion, however to quickly summarize it, we take a string such as,

0A0BY1988487665 (obviously a random example for simple explanation)

Convert it into a list,

['0', 'A', '0', 'B', 'Y', '1', '9', '8', '8', '4', '8', '7', '6', '6', '5']

Parse it into,

[0, 'A', 0, 'B', 19884, 87665]

And finally, decode it. This is not a sane example, but gets the basic intuition of decoding across.

Decoding generic LZ encoded strings is extremely direct, and, we just keep appending an empty list appropriately (appending unencoded entities directly, appending encoded entities according to their P and L entries)

3. Results

My final encoded text size was 700 kB (84% compression)

My final decoded text size was 832 kB (same as the input one)

Additionally, the decoded text exactly matched the input text. My encoding runtime was around 3 hours, decoding runtime was nominally zero.

I could've gotten better results if I'd worked on the problem a bit more, and considered other compression techniques or minor modifications over my own technique, and could have also reduced the encoding runtime, however the deadlines constrained further exploration. One issue is that my entire strategy is exactingly sequential, and dependent on

things being done sequentially. If someone randomly wanted to decode just a segment of the input text, there might be some issues. Again resolution of these issues, along with eradication of such issues were limited by the deadlines.